Criptografia i Seguretat

Laboratori

Curs 2024-25

UAB

Laboratori 2:

Hash: Filtres Bloom i eines de força bruta

(Data: 2025-04-19)

David Morillo Massagué

Adrià Muro Gómez

Taula de continguts

Introducció

L'autenticació d'usuaris mitjançant contrasenyes és una tècnica àmpliament utilitzada per protegir l'accés als sistemes informàtics. Tot i la seva simplicitat i rapidesa, aquest mecanisme també presenta vulnerabilitats importants, especialment quan les contrasenyes es filtren o són emmagatzemades sense protecció adequada.

Aquesta pràctica aborda dos aspectes fonamentals relacionats amb la seguretat de contrasenyes: la detecció eficient de contrasenyes compromeses mitjançant filtres de Bloom, i les tècniques per trencar hashos de contrasenyes, amb l'objectiu de comprendre millor les debilitats dels sistemes d'autenticació i els riscos associats a un mal emmagatzematge de contrasenyes.

Objectius

Part I: Filtres de Bloom

- Comprendre el funcionament intern dels filtres de Bloom i el seu ús en contextos de seguretat.
- Implementar un filtre de Bloom configurable en mida i nombre de funcions hash, amb funcionalitats de serialització.
- Analitzar l'impacte del nombre de funcions hash sobre la probabilitat de falsos positius, tant de forma teòrica com experimental.
- Comparar l'eficiència del filtre de Bloom amb altres estructures de dades, com conjunts o llistes.
- Optimitzar el rendiment del filtre mitjançant la tècnica de doble hashing.
- Crear un filtre de Bloom per tot un conjunt massiu de contrasenyes, escollint els paràmetres òptims segons una taxa de falsos positius màxima permesa.

Part II: Hash cracking

- Comprendre la necessitat d'emmagatzemar contrasenyes en forma de hash i els riscos de seguretat associats a filtracions.
- Aplicar diferents tècniques per trencar hashos: atacs de força bruta, atacs de diccionari i atacs contextuals.

- Familiaritzar-se amb eines com *John the Ripper* i *haiti* per a la identificació i desxifrat de funcions hash.

- Analitzar l'impacte de la mida de l'alfabet i la longitud de la contrasenya sobre la viabilitat d'un atac.

- Experimentar amb l'ús de *salts* i alteracions habituals en la construcció de contrasenyes per simular escenaris reals de seguretat.

Bloom filters

Els filtres de Bloom són estructures de dades probabilístiques que permeten comprovar si un element pertany a un conjunt de manera eficient i amb un consum reduït de memòria. Tot i que poden generar falsos positius, mai produeixen falsos negatius, fet que els fa útils en contextos com la detecció de contrasenyes filtrades. En aquesta pràctica es treballa amb una base de dades real de 1.4 mil milions de credencials en clar per implementar i analitzar un filtre de Bloom en diferents escenaris.

Exercici 1: Implementació d'un filtre de Bloom

S'ha implementat una classe BloomFilter que permet:

- **Inicialitzar un filtre amb una mida i un nombre determinat de funcions hash:**
  - __init__(self, size, num_hashes) → Els atributs que guardarà la classe seran el tamany, el nombre de hashes per cada element i l'array de bits inicialitzats a 0.

- **Afegir elements al filtre:**
  - add(self, item) → Calcula els diferents valors hash de l'element i converteix a 1 les posicions corresponents als hashes de l'element en l'*array* de bits.

- **Comprovar la presència d'elements:**
  - check(self, item) → Revisa totes les posicions dels diferents hashes de l'element i només si totes són 1 retornarà *True*.

- **Desar i carregar el filtre des de disc:**
  - save(self, filename) → Guarda el filtre a disc
  - load(filename) → Carrega el filtre des de disc

```
class BloomFilter:
    def __init__(self, size, num_hashes):
        self.size = size
        self.num_hashes = num_hashes
        self.bit_array = bitarray.bitarray(size)
        self.bit_array.setall(0)

    def _hashes(self, item):
        return [mmh3.hash(item, i) % self.size for i in range(self.num_hashes)]

    def add(self, item):
        for hash_value in self._hashes(item):
            self.bit_array[hash_value] = 1
```

```
    def check(self, item):
        return all(self.bit_array[hash_value] for hash_value in
self._hashes(item))

    def save(self, filename):
        with open(filename, 'wb') as f:
            pickle.dump(self, f)

    @staticmethod
    def load(filename):
        with open(filename, 'rb') as f:
            return pickle.load(f)
```

Exercici 2: Impacte del nombre de funcions hash

Per resoldre aquest exercici s'ha seleccionat un subconjunt de contrasenyes i s'han generat diversos filtres de Bloom amb el mateix tamany però amb diferent nombre de funcions hash k. Per cada k, s'ha calculat:

- **La taxa teòrica de falsos positius:**

  Per calcular la taxa teòrica de falsos positius s'ha utilitzat la fórmula corresponent al comportament d'un filtre de Bloom, la qual depèn dels paràmetres següents: **n**, el nombre d'elements inserits al filtre; **m**, la mida del filtre en bits; i **k**, el nombre de funcions de hash utilitzades.

  Aquesta fórmula permet estimar la probabilitat que el filtre indiqui erròniament la presència d'un element no inserit.

  ```
  def false_positive_rate(n, m, k):

      return (1 - np.exp(-k * n / m)) ** k
  ```

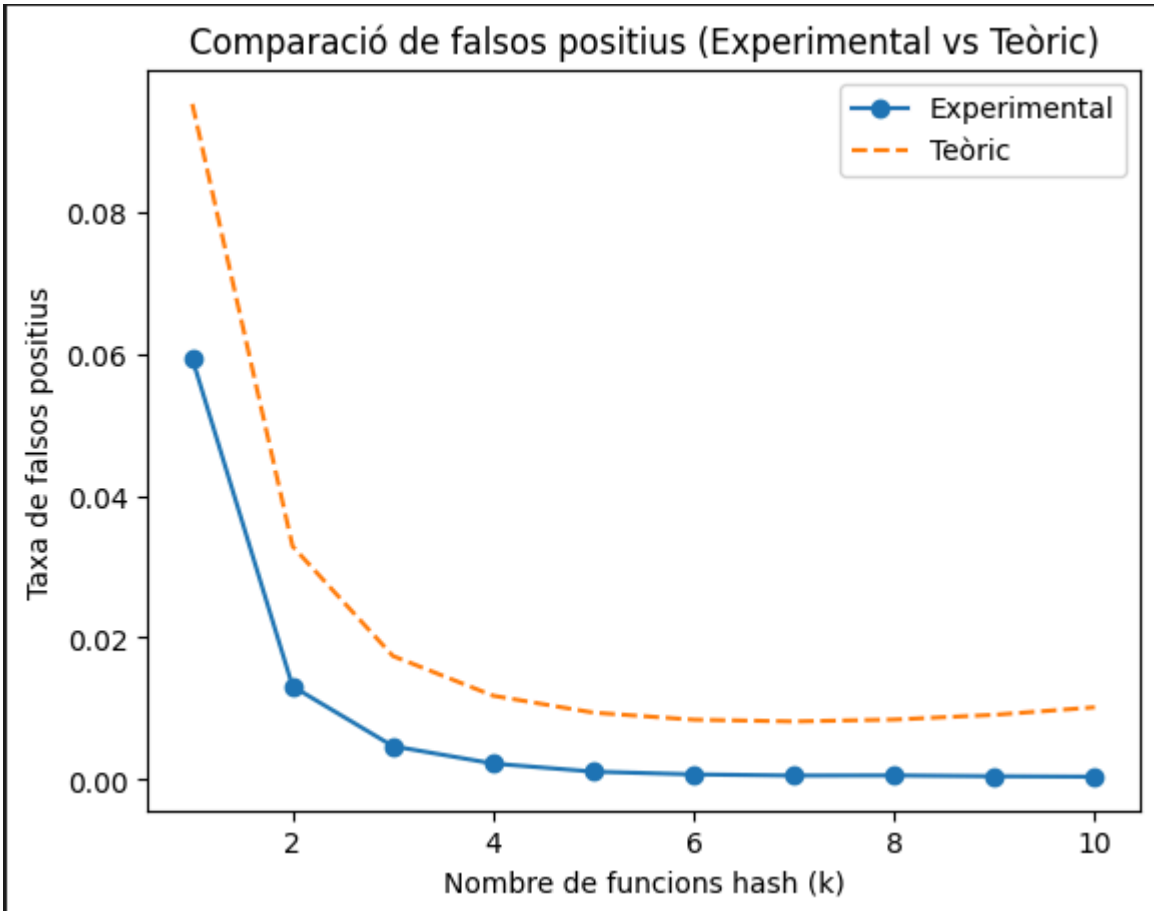
- **La taxa experimental (amb contrasenyes no afegides al filtre):**

  Per calcular la taxa de falsos positius experimental, s'ha generat un conjunt de contrasenyes artificials que no es troben al filtre. A continuació, s'ha iterat sobre aquest conjunt verificant si cada contrasenya era identificada com a present pel filtre. Cada vegada que el filtre ha retornat un resultat positiu incorrecte, s'ha incrementat un comptador.

Finalment, la taxa s'ha obtingut dividint el nombre de falsos positius entre el nombre total de contrasenyes analitzades.

```
test_passwords = [f"fakepass{i}" for i in range(n)]

 false_positives = sum(1 for p in test_passwords if bf.check(p))

 false_positive_rates_exp.append(false_positives / n)
```

La gràfica generada mostra l'evolució de la probabilitat de falsos positius en funció del nombre de funcions de hash (**k**), tant a nivell teòric com experimental.



S'observa que ambdós valors segueixen una trajectòria molt similar, fet que valida la correcció de la implementació i el comportament esperat del filtre de Bloom. Tot i això, la taxa experimental es troba lleugerament per sota de la teòrica, especialment en alguns valors de **k**.

Aquest desfasament pot atribuir-se al fet que l'experiment s'ha realitzat amb un subconjunt limitat de contrasenyes, molt menor que la mida total del dataset. És raonable, doncs, que les taxes obtingudes experimentalment siguin una mica més baixes, ja que amb menys col·lisions en el filtre també disminueix la probabilitat de falsos positius.

Això confirma que els resultats observats són coherents amb el funcionament probabilístic del filtre.

Exercici 3: Selecció òptima de paràmetres

S'ha implementat una funció que permet calcular els paràmetres òptims d'un filtre de Bloom a partir de dues variables d'entrada: **n**, el nombre d'elements que s'espera inserir, i **p**, la taxa de falsos positius desitjada.

```
def calcular_parametres_bloom(n, p):

    m = - (n * math.log(p)) / (math.log(2) ** 2)

    k = (m / n) * math.log(2)

    return int(m), int(k)
```

La funció es basa en les fórmules matemàtiques estàndard dels filtres de Bloom per determinar:

- **m** → La mida òptima del filtre (en bits),
- **k** → El nombre òptim de funcions de hash

La funció retorna aquests dos valors convertits a enters, ja que el filtre de Bloom requereix una mida discreta i un nombre sencer de funcions hash. Aquest càlcul permet configurar el filtre de manera eficient, maximitzant l'eficiència de l'espai i minimitzant la probabilitat de falsos positius per al volum de dades previst.

Exercici 4: Comparativa amb una estructura no probabilística

S'ha dut a terme una comparativa experimental entre dues estructures de dades per comprovar la presència de contrasenyes: un **filtre de Bloom**, que és una estructura probabilística, i un **conjunt (set) de Python**, que és una estructura exacta i no probabilística.

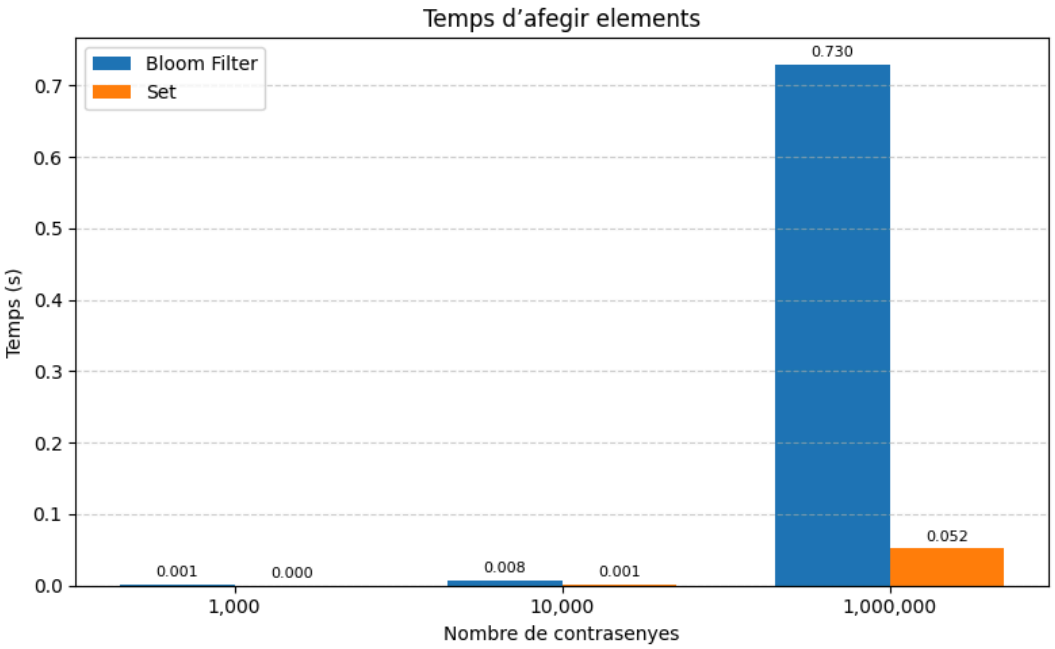
La comparació s'ha fet tenint en compte quatre mètriques essencials:

1. **Temps d'afegir elements**
2. **Temps de comprovació de pertinença**
3. **Mida de l'estructura de dades**
4. **Taxa de falsos positius**

Els experiments s'han repetit per a conjunts de 3 mides diferents: 1.000, 10.000 i 1.000.000 contrasenyes. S'han seleccionat subconjunts del dataset de contrasenyes per cada cas.

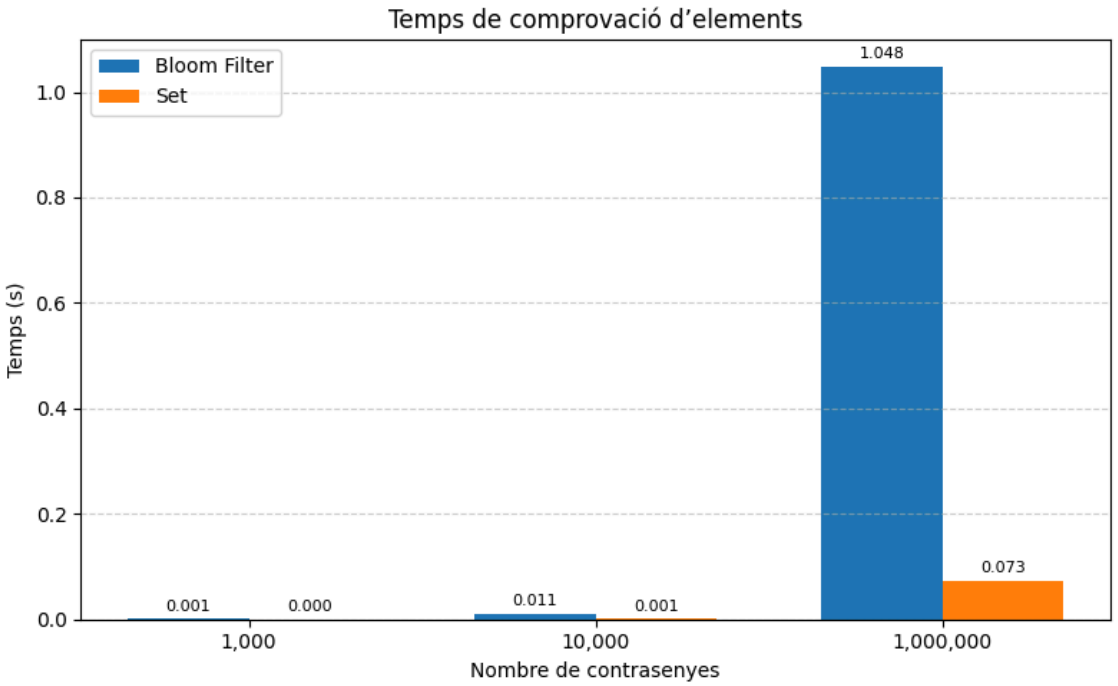
S'han generat quatre gràfiques de barres comparant les dues estructures (set i filtre de Bloom) per cada mètrica. L'eix X mostra la mida del conjunt de dades (nombre de contrasenyes), i l'eix Y representa el valor mesurat per la mètrica corresponent.

1. **Temps d'afegir elements**:



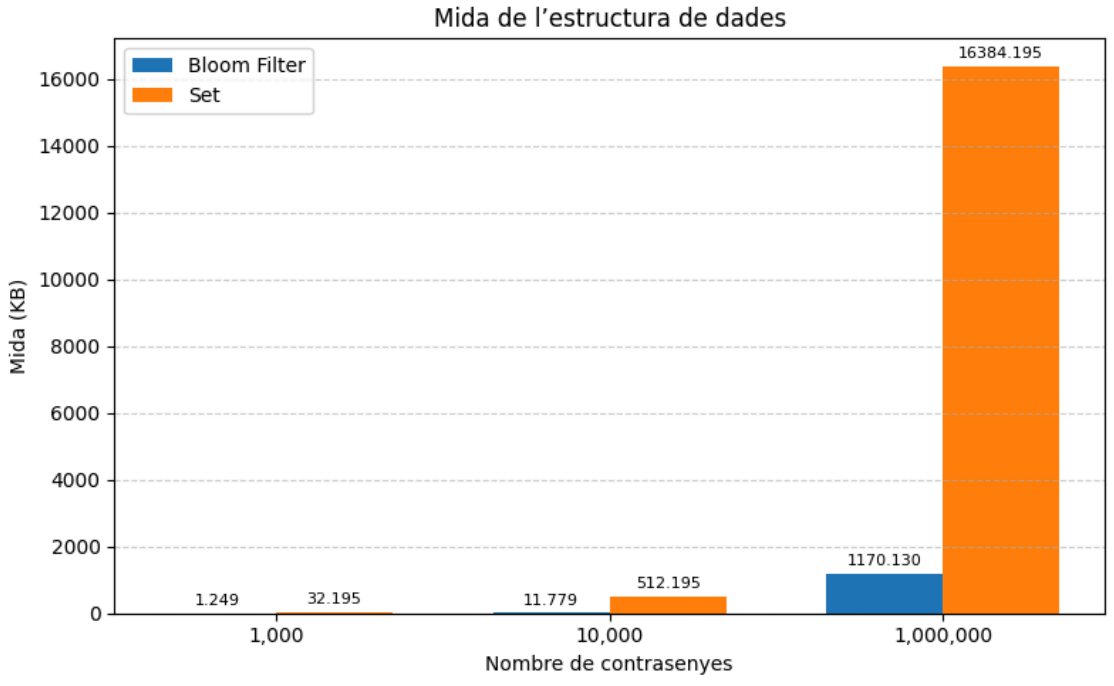
El set ha estat sistemàticament més ràpid que el filtre de Bloom. Això es deu al fet que els conjunts de Python estan altament optimitzats i realitzen la inserció en temps mitjà constant, mentre que el filtre de Bloom ha de calcular diverses funcions hash per cada element, fet que incrementa el temps d'execució.

2. **Temps de comprovació de pertinença**:



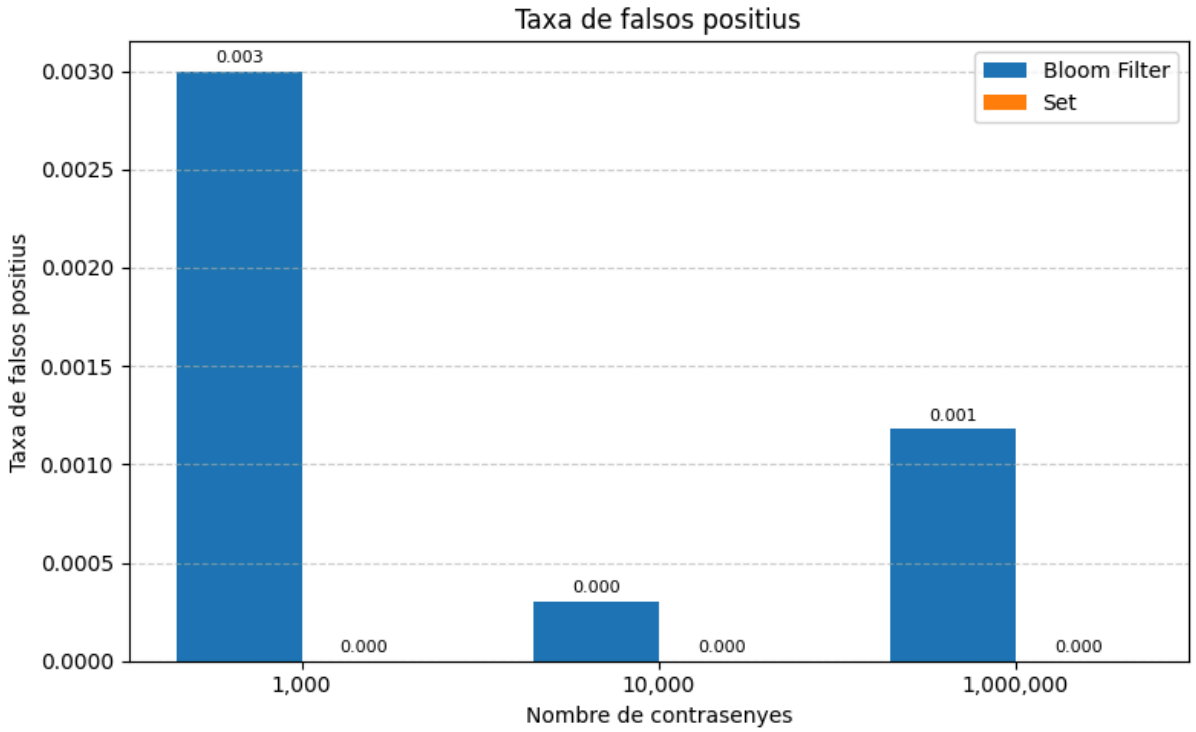
Igual que en el cas anterior, el set ha superat en velocitat al filtre de Bloom. Novament, aquest comportament és esperable, ja que el set fa una cerca directa per hash, mentre que el filtre de Bloom requereix calcular k valors de hash i accedir a múltiples posicions del seu array de bits.

3. **Mida de l'estructura de dades**:



Aquesta és la principal avantatge del filtre de Bloom. La seva mida és molt inferior a la del set, especialment a mesura que augmenta el nombre de contrasenyes. Mentre que el set ha de guardar tots els objectes i la seva informació, el filtre de Bloom només manté un vector de bits, cosa que permet una gran eficiència en memòria.

4. **Taxa de falsos positius**:



Com era d'esperar, el set no presenta cap fals positiu, mentre que el filtre de Bloom sí que ho fa, tot i que dins dels marges esperats segons la configuració.

Els resultats obtinguts són coherents amb el comportament teòric de les estructures:

- El set és més ràpid en operacions d'afegir i consultar, però consumeix molta més memòria.
- El filtre de Bloom és ideal quan la memòria és un recurs limitat i podem tolerar una certa probabilitat de falsos positius.

Exercici 5: Optimització amb doble hashing

Amb l'objectiu de reduir el cost computacional associat al càlcul de múltiples funcions hash, s'ha implementat el mètode de doble hashing. Aquest consisteix a calcular només dues funcions hash independents i generar la resta de valors mitjançant una combinació lineal:

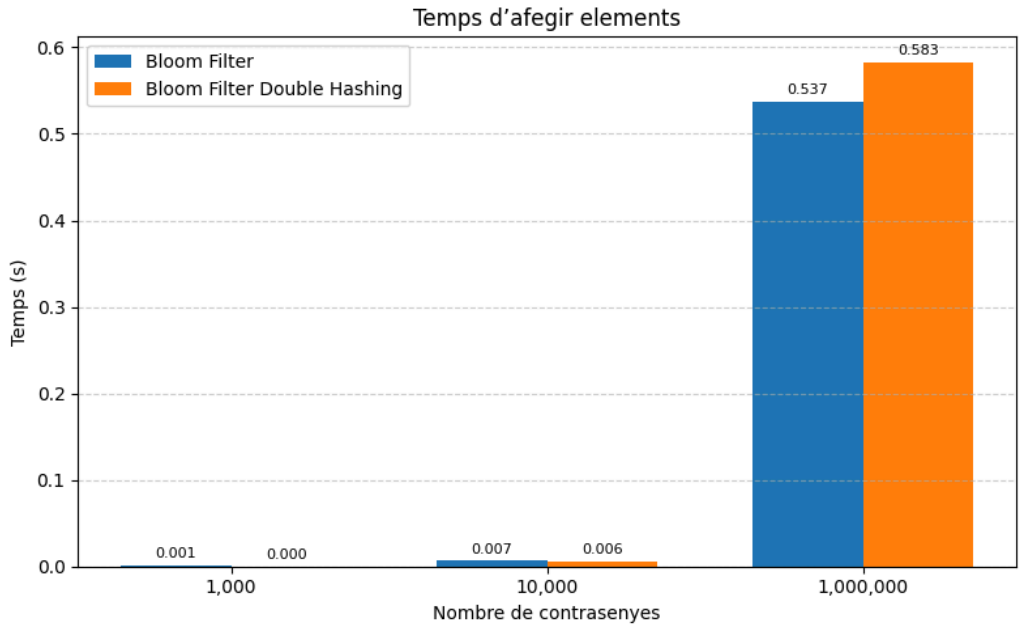
```
def _hashes(self, item):

        h1 = mmh3.hash(item, 42) % self.size

        h2 = mmh3.hash(item, 84) % self.size

        return [(h1 + i * h2) % self.size for i in range(self.num_hashes)]
```

Aquesta tècnica teòricament permet reduir de forma notable el temps d'execució, ja que evita fer *k* crides a funcions hash independents i substitueix-les per operacions aritmètiques simples. Tot i aquesta optimització, es manté l'eficàcia del filtratge, amb una taxa de falsos positius molt propera a la del mètode clàssic, la qual cosa la converteix en una solució eficient tant en rendiment com en qualitat.

Exercici 6: Comparativa entre funcions independents i doble hashing

S'han comparat les dues implementacions del filtre de Bloom utilitzant els mateixos subconjunts de dades i mètriques que a l'exercici 4. En aquesta comparativa, hem analitzat quatre mètriques principals: el temps per afegir elements al filtre, el temps per comprovar elements, la mida de l'estructura de dades i la taxa de falsos positius, per tal d'avaluar les diferències entre les dues tècniques.

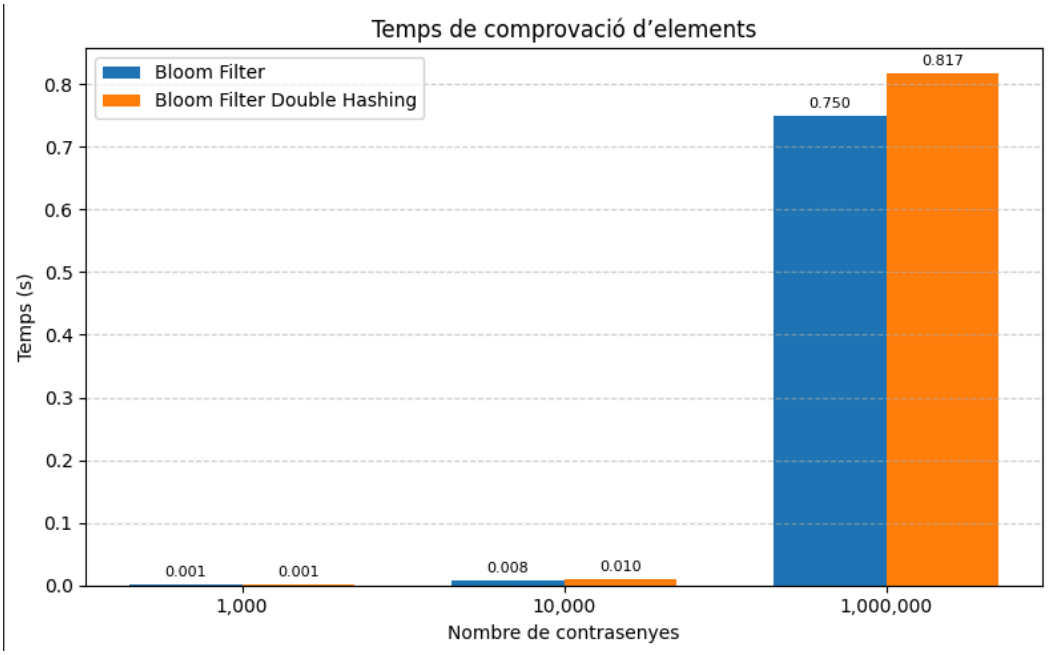
Temps d’afegir elements



La primera gràfica mostra el temps necessari per afegir elements a la estructura del filtre de Bloom. Els resultats indiquen que el filtre de Bloom amb funcions hash independents és més ràpid en afegir elements que el filtre amb doble hashing.

Això és comprensible, ja que el doble hashing implica càlculs addicionals en cada inserció, fet que provoca una lleugera penalització en el temps d’afegir. No obstant, cal recordar que el doble hashing no necessita fer k crides al hash, només dues crides.

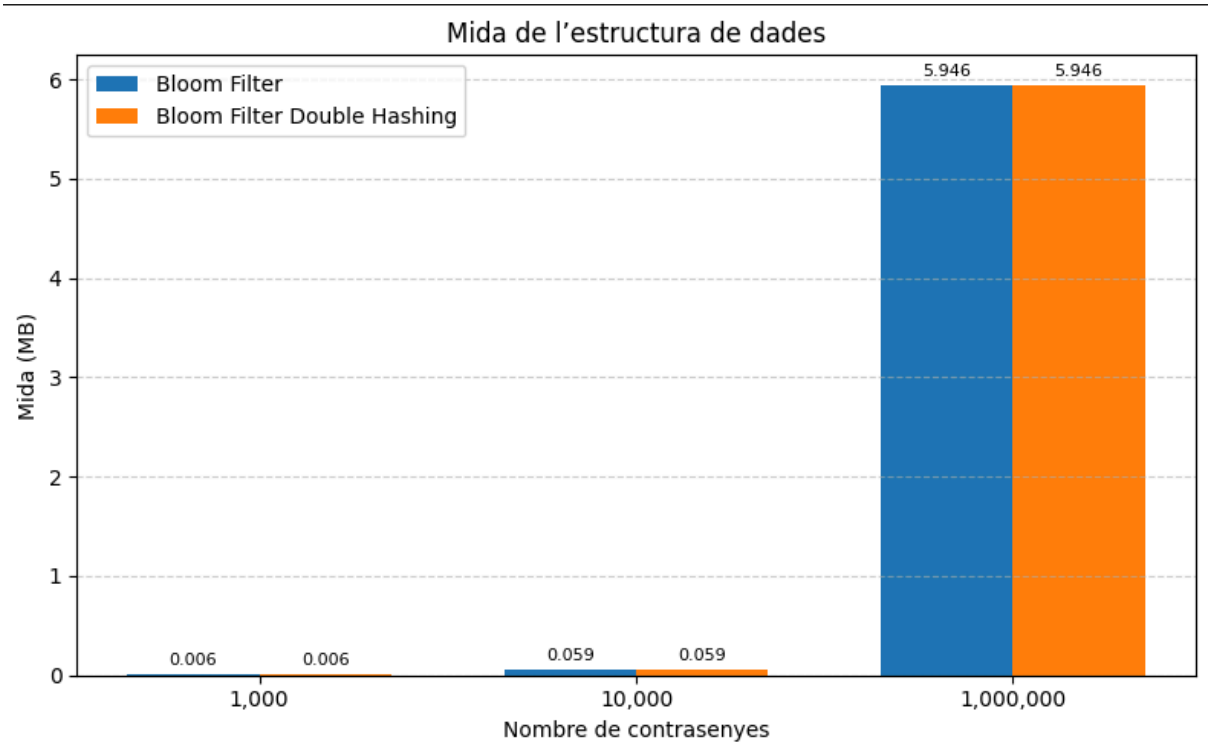
Temps de comprovació d’elements



En la segona gràfica, observem el temps que es necessita per comprovar la presència d'un element al filtre de Bloom. En aquest cas, els resultats mostren que el temps de comprovació és lleugerament més ràpid per al filtre amb funcions hash independents en comparació amb el doble hashing.

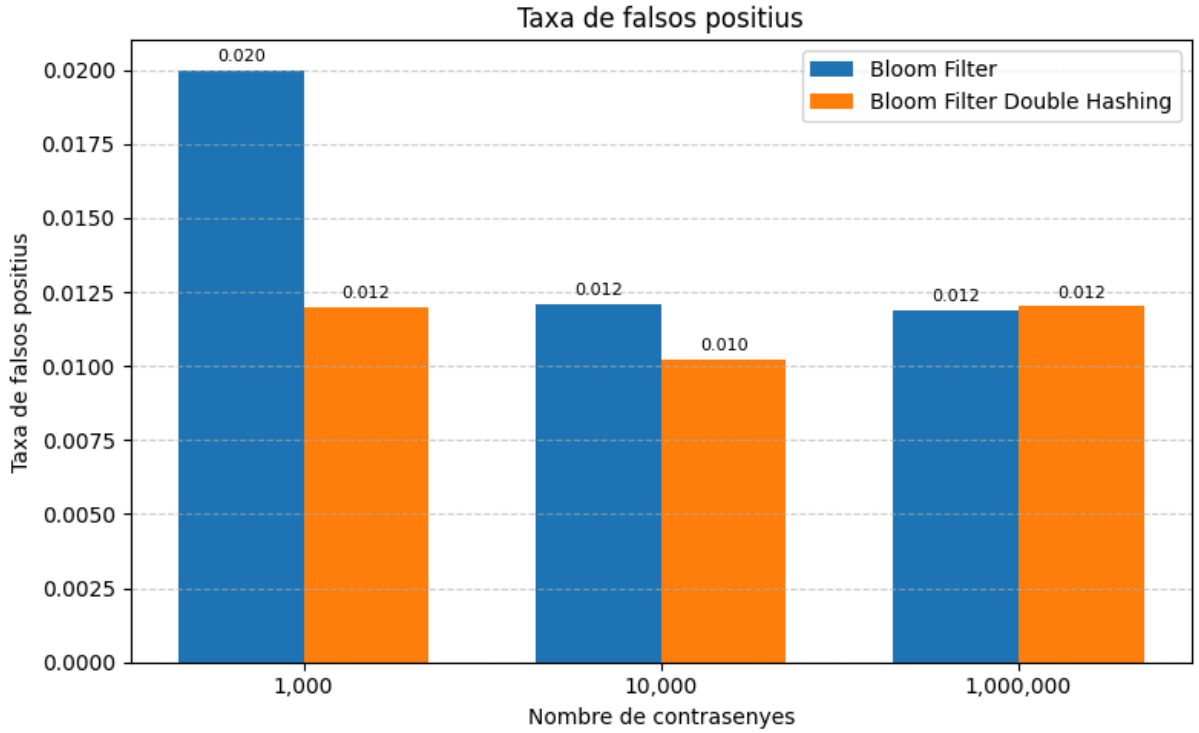
Això es deu a la simplicitat del procés de comprovació en el cas de funcions hash independents, que no requereixen la combinació de dues funcions hash com en el doble hashing.

Mida de l'estructura de dades



La tercera gràfica compara la mida de les estructures de dades per a cada tipus de filtre de Bloom. Els resultats mostren que les dues implementacions (funcions hash independents i doble hashing) ocupen la mateixa quantitat d'espai de memòria, amb una mida similar per a ambdues tècniques. Això indica que, malgrat les diferències en el temps de processament i la taxa de falsos positius, ambdues implementacions utilitzen el mateix espai per a emmagatzemar la informació del filtre de Bloom.

Taxa de falsos positius



Finalment, la quarta gràfica mostra la taxa de falsos positius per a cada filtre de Bloom. Els resultats obtinguts indiquen que el filtre de Bloom amb doble hashing presenta una taxa de falsos positius més baixa que el filtre amb funcions hash independents. Això es deu a la manera com el doble hashing distribueix els elements dins del filtre de manera més uniforme, reduint les probabilitats de col·lisions i, per tant, els falsos positius.

Exercici 7: Aplicació al dataset complet

En aquest exercici, s'ha creat un filtre de Bloom per al conjunt de dades de contrasenyes utilitzant la millor configuració possible, amb una taxa de falsos positius del 5%. A continuació, es responen les preguntes específiques a partir dels resultats obtinguts:

1. **Quina versió de la implementació heu utilitzat?**
   S'ha utilitzat la implementació del filtre de Bloom amb funcions hash independents, ja que es sorprenentment ofereix un millor rendiment en quant a temps que el doble hashing (revisar comparativa exercici 6).

2. **Quins són els paràmetres (mida i nombre de funcions hash) del filtre?**
   Els paràmetres òptims per al filtre de Bloom, obtinguts mitjançant els càlculs realitzats, són:

   - Mida del filtre de Bloom: 8.654.354.012 bits (aproximadament 1,08 GB)

- Nombre de funcions hash: 4

3. **Quants elements heu afegit al filtre?**
   S'han afegit un total de 1.387.977.993 contrasenyes al filtre de Bloom.

4. Compareu l'espai necessari per emmagatzemar el filtre amb la mida del conjunt de dades.

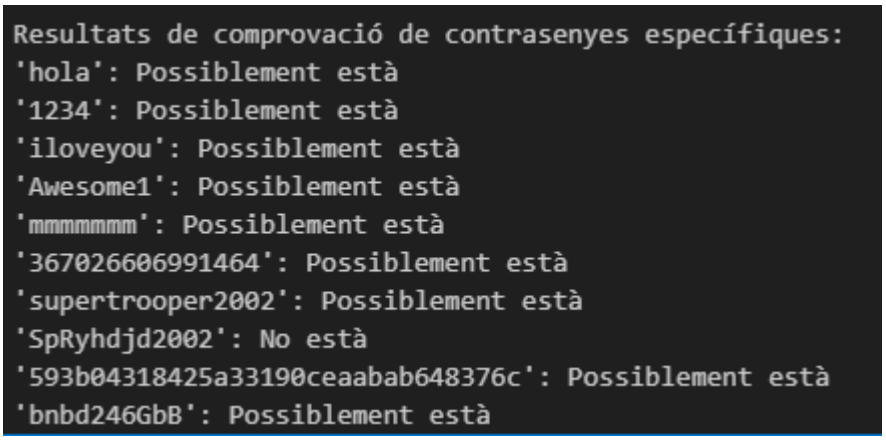
   - Mida del filtre de Bloom: 1031,68 MB
   - Mida de la llista de contrasenyes: 11.252,79 MB

   Com es pot observar, el filtre de Bloom utilitza una quantitat menor d'espai en comparació amb la mida total del conjunt de dades. El filtre de Bloom ocupa només una petita fracció de l'espai que es necessita per emmagatzemar les contrasenyes originals en la seva totalitat.

5. Quant temps ha estat necessari per afegir totes les contrasenyes al filtre?

   El temps total necessari per afegir les 1.387.977.993 contrasenyes al filtre de Bloom ha estat de 1718.348432 segons, o aproximadament 28.63 minuts. Aquest temps es refereix al temps total de processament per afegir tots els elements al filtre de Bloom.

6. Indiqueu quines de les contrasenyes següents s'han filtrat (és a dir, es troben al filtre).



   Les contrasenyes que possiblement estan al filtre, segons la comprovació realitzada, són:

   - 'hola': Possiblement està

- '1234': Possiblement està
- 'iloveyou': Possiblement està
- 'Awesome1': Possiblement està
- 'mmmmmmm': Possiblement està
- '367026606991464': Possiblement està
- 'supertrooper2002': Possiblement està
- 'SpRyhdjd2002': No està
- '593b04318425a33190ceaabab648376c': Possiblement està
- 'bnbd246GbB': Possiblement està

Totes les contrasenyes proporcionades menys SpRyhdjd2002, que no està, possiblement es troben al filtre, fet que mostra com el filtre de Bloom pot identificar eficientment si un element ha estat afegit, tot i que no pot garantir una resposta exacta (degut a la natura dels falsos positius).

Hash cracking

Amb els coneixements adquirits fins ara, sabem que un pas fonamental per guardar contrasenyes de manera segura és aplicar una funció *hash*. Això impedeix que un atacant pugui veure les contrasenyes en clar només accedint al fitxer on s'emmagatzemen. Tot i així, a mesura que la tecnologia avança, també ho fa la potència dels ordinadors, fins i tot dels convencionals. Per això, alguns algoritmes de *hashing* que abans es consideraven segurs, ara ja no ho són, perquè és massa fàcil per a un atacant provar totes les combinacions possibles (**força bruta**).

Les funcions *hash* tenen el que s'anomena resistència a la preimatge: és molt fàcil calcular el *hash* d'un valor, però gairebé impossible fer el camí invers, és a dir, descobrir quin valor original el va generar. Però si l'atac es fa a petita escala i amb prou iteracions, es poden arribar a descobrir els valors originals. Això es fa provant molts valors diferents fins que el seu *hash* coincideix amb el que es vol desxifrar.

Aquesta és precisament la tècnica que utilitza l'eina **John the Ripper** en sistemes Linux. Tal com indica el seu nom, "destripa" hashes utilitzant força bruta per intentar trobar el valor original. En aquesta segona part de la pràctica posarem en pràctica aquesta eina per a tal de desxifrar hashes donats.

Per a fer servir l'eina mencionada anteriorment, la vam instalar amb la comanda:

```
sudo snap install john-the-ripper
```

Exercici 8: Atacs de força bruta

Vam crear un arxiu *hashes.txt* i vam afegir els hashes que haviem de desxifrar:

```
dakur@lg:~/Documents$ cat hashes.txt
h4_1:6ea9ab1baa0efb9e19094440c317e21b
h4_2:8e296a067a37563370ded05f5a3bf3ec
h4_3:54fe976ba170c19ebae453679b362263
h4_4:6562c5c1f33db6e05a082a88cddab5ea
h4_5:7c590f287acefdd3ea84a7678f1e907b
h4_6:6593a1651adf82783394195112e73aac
h4_7:a388742c988cb1b8d9a304db528cf71d
h4_8:047d8415eec2dcec989c77d531535531
h4_9:b87262873e28e7589c15c5e467e9c39a
h4_10:42005f9a3f3a28aabe4883bb7a60ec0a
h4_11:1f99c8a687de5b829addfce79383827a
h4_12:1d1803570245aa620446518b2154f324
```

Amb el format {nom}:{hash} podrem veure a quin hash pertany cada valor en clar, a mesura que es vagin desxifrant, com veurem més a continuació. Hem fet servir aquest format també amb la resta d'exercicis.

Per a un millor ús de l'eina John the Ripper, és millor especificar-li el tipus de funció hash que volem que utilitzi per a desxifrar el hash. Per a això s'ha fet servir l'eina Hash Analyzer

que, donada l'output d'un hash qualsevol, indica el tipus de funció que s'ha utilitzat per a donar tal sortida. Donat que en aquest exercici els hashes tenien el matiex format, s'ha assumit que els 12 s'havien encriptat amb la mateixa funció. Com observem en l'imatge anterior, la sortida consta de 32 caràcters hexadecimals (128 bits). L'eina Hash Analyzer ha determinat que es tractava d'un xifrat **MD5 o MD4**.

A l'enunciat es menciona que *"aquestes (preimatges) estan formades només per dígits",* cosa que va reduir significativament el nombre de combinacions que haviem de provar.

Un cop sabíem l'algoritme que s'havia fet servir, vam executar la comanda:

```
john-the-ripper --format=raw-md5 --incremental=Digits hashes.txt
```

Aquesta comanda feia servir dígits com a entrada (--incremental) i els transformava amb la funció **MD5** (--format) per a fer els atacs de forma bruta. La sortida que ens va donar va ser la següent:

```
dakur@lg:~/Documents$ john-the-ripper --format=raw-md5 --incremental=Digits hashes.txt
Using default input encoding: UTF-8
Loaded 12 password hashes with no different salts (Raw-MD5 [MD5 512/512 AVX512BW 16x3])
Warning: no OpenMP support for this hash type, consider --fork=8
Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for status
29               (h4_1)
25               (h4_2)
4567             (h4_4)
4999             (h4_3)
699999           (h4_5)
645698           (h4_6)
1011111111       (h4_10)
81234567         (h4_8)
83456789         (h4_7)
123456789101     (h4_11)
1098765432       (h4_9)
121212121212     (h4_12)
12g 0:00:01:46 DONE (2025-04-18 19:42) 0.1127g/s 53075Kp/s 53075Kc/s 83046KC/s 121219931103..121214111033
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed.
```

Com a observació d'aquesta primera execució, podem veure com les primeres preimatges que s'han anat trobant han sigut els nombres més petits. Amb això es comprova com aquesta eina comprova els valors amb un ordre definit: de més petits a més grans. Naturalment, fa que com és gran sigui el nombre en clar, més iteracions cal fer per a desxifrar el seu hash, si seguim aquest ordre, però menys segur sigui, si recorrem aquests digits a l'inrevés! Veiem que alguns nombres s'han endevinat abans que d'altres, com el cas del 25 i el 29. Deduïm que això és causat pel paral·lelisme que fa servir l'algoritme, i que no tots els fils de la CPU iteren exactament a la vegada.

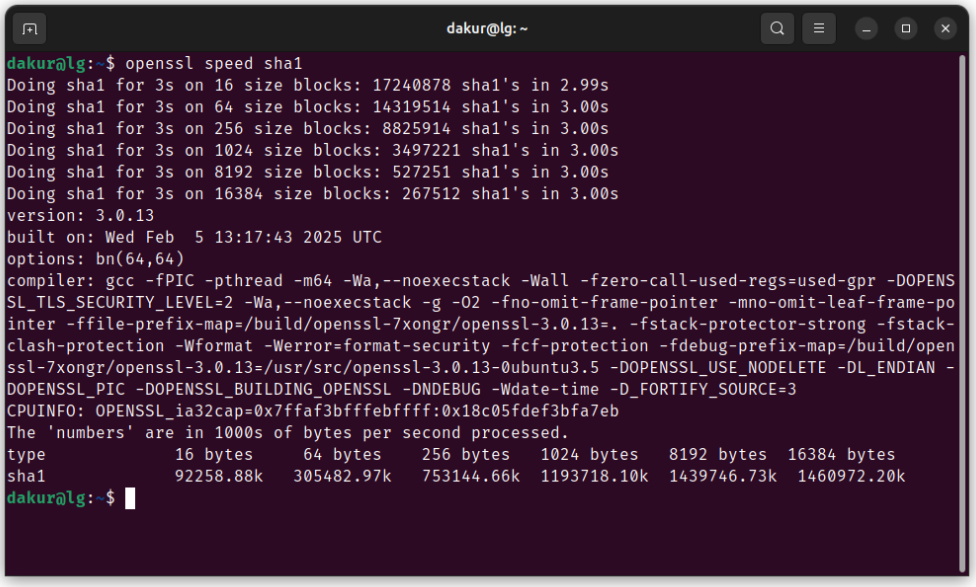
Exercici 9: Atacs de força bruta - mida de l'alfabet

Per a una contrasenya de longitud **n**, el nombre màxim de proves amb força bruta és igual al nombre total de combinacions possibles, que depèn de la mida de l'alfabet. Com que a l'exercici anterior s'especificava que l'alfabet només era numèric, la mida de l'alfabet és de **a=10** ({0-9}). La contrasenya en clar més llarga de l'exercici anterior és "121212121212", amb una longitud de **n=12** dígits. El nombre de combinacions amb aquestes característiques és de $\mathbf{a^n = 10^{12} = 1.000.000.000.000}$ combinacions. Amb una contrasenya

alfanumerica, l'alfabet seria de mida a=26 lletres + 10 dígits → a=36, i la formula seria: $\mathbf{a^n = 36^n}$. Per cadascuna de les longituds de l'exercici anterior:

Contrasenya	Longitud	Combinacions amb a=10	Combinacions amb a=36
h4_1	2	100	1296
h4_2			
h4_3	4	10.000	1679616
h4_4			
h4_5	6	1.000.000	2176782336
h4_6			
h4_7	8	100.000.000	$2.82111*10^{12}$
h4_8			
h4_9	10	10.000.000.000	$3.65616*10^{15}$
h4_10	11	100.000.000.000	$1.31622*10^{17}$
h4_11	12	1.000.000.000.000 = 10*12	$4.73838*10^{18}$
h4_12			

Veiem com, al tenir un alfabet més gran, les combinacions creixen molt més ràpid i, per tant, més segura és la contrasenya contra atacs de força bruta.



Amb la comanda `openssl speed sha1` observem la potència que té el processador de la màquina que estem fent servir: és capaç de comprovar gairebé 3 milions de hash de 64 blocs en 1 segon.

Això vol dir que, si fem servir una contrasenya de **8 caràcters (64 bits)**:

- Si fem servir només dígits, es poden recórrer les 100.000.000 combinacions **34 segons**
- Si fem servir una contrasenya alfanumerica, es poden recórrer les $2.82111*10^{12}$ combinacions en poc més de **6,84 dies**

Donats aquests resultats podem apreciar la gran diferència entre fer servir un diccionari, o un altre. El simple fet d'incloure catàcters a la contrasenya fa que un atacant trigui potencialment **1,7 milions de vegades més** (en relació al temps) a trobar la contrasenya per força bruta.

Exercici 10: Atacs de diccionari

Per a aquest exercici s'ha fet servir el diccionari Rockyou.txt que conté més de 32 milions de contrasenyes en clar d'usuaris de la companyia RockYou al 2009.

Les funcions comprovats per als hashes donats han sigut els següents:

```
formats = ["Raw-MD5", "md5crypt", "Raw-SHA256", "Raw-SHA512", "sha256crypt"]
```

Amb la següent comanda s'han comprovat per força bruta les contrasenyes dels hashes donats:

```
john-the-ripper --format={format} --wordlist=rockyou.txt hashes2.txt
```

Hem fet servir l'argument --wordlist en aquest cas, per a que faci les comprovacions utilitzant com a entrada les paraules en clar de la llista especificada.

Un cop processades i desxifrades, amb el següent script de python hem obtingut la sortida següent:

```
for f in formats:
    print(f"Format: {f}")
    !john-the-ripper --show --format={f} hashes2.txt
    print("#" * 40)
```

```
Format: Raw-MD5
h6_1:1996gis

1 password hash cracked, 0 left
########################################
Format: md5crypt
h6_4:Unsecure321

1 password hash cracked, 0 left
########################################
Format: Raw-SHA256
h6_2:criptonomicon

1 password hash cracked, 0 left
########################################
Format: Raw-SHA512
0 password hashes cracked, 1 left
########################################
Format: sha256crypt
h6_5:Unsecure321

1 password hash cracked, 0 left
########################################
```

Veiem que s'han pogut desxifrar totes les contrasenyes, excepte la h6_3. Deduïm que això és degut a que no hem pogut trobar un diccionari de filtracions amb aquesta contrasenya en clar, o que no hem trobat la funció hash correcta per a aquest cas.

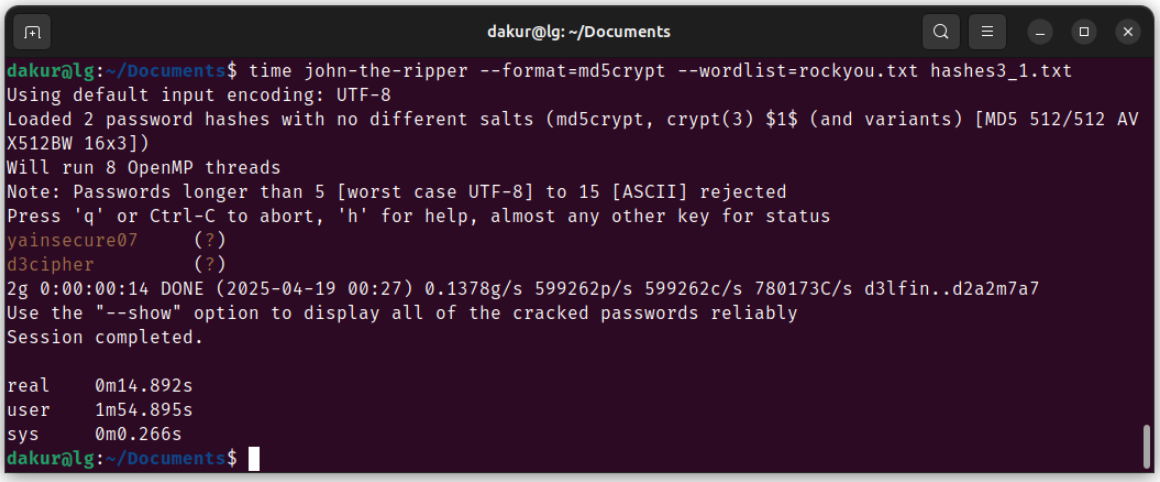
El que ha trigat més en desxifrar-se és el sha256crypt que ha trigat 570 segons, cosa esperada per la seva naturalesa més segura i resistent als atacs de força bruta.

Exercici 11: L'ús de sal

Per a aquest exercici hem executat les següents comandes:

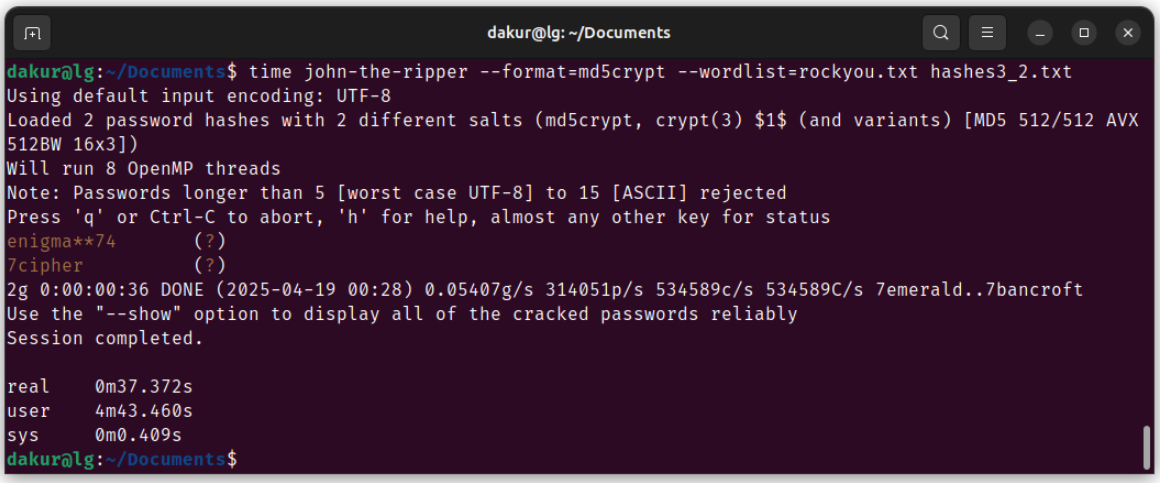
```
time john-the-ripper --format=md5crypt --wordlist=rockyou.txt hashes3_1.txt
```

Hem fet servir altre cop el diccionari Rockyou.txt. Hem determinat, segons els outputs donats, que la funció feta servir havia sigut MD5-Crypt, ja que els hash començaven per "1". La comanda ha produït la següent sortida:



time john-the-ripper --format=md5crypt --wordlist=rockyou.txt hashes3_2.txt

Que ha produït la següent sortida:



Podem concloure que l'ús de salts diferents per contrasenya (2n cas) incrementa significativament el temps d'un atac de diccionari, ja que el hash s'ha de recalcular per cada combinació de contrasenya i salt.

Quan diverses contrasenyes comparteixen el mateix salt (1r cas), es pot reutilitzar el mateix diccionari prèviament hashejat amb aquest salt, fent l'atac molt més eficient.

L'increment de temps no és lineal, però és proporcional a la diversitat de salts: cada nou salt obliga a repetir tot l'atac.

Conclusions

Aquesta pràctica ha permès aprofundir en dos àmbits fonamentals de la seguretat de contrasenyes: la detecció preventiva mitjançant estructures eficients com els filtres de Bloom i les tècniques ofensives d'atac per trencar hashos.

A través de la primera part, hem comprovat com els filtres de Bloom poden representar una solució eficient per detectar contrasenyes compromeses en grans conjunts de dades, gràcies a la seva capacitat de consum reduït de memòria i temps. Tot i que introdueixen falsos positius, hem vist que aquesta limitació es pot controlar ajustant els paràmetres del filtre i aplicant tècniques d'optimització com el doble hashing.

D'altra banda, en la segona part, s'ha evidenciat la importància d'emmagatzemar les contrasenyes de forma segura utilitzant funcions hash robustes i mecanismes com l'ús de *salts*. Els exercicis han permès entendre de manera pràctica la vulnerabilitat de contrasenyes febles o predecibles, i com eines com *John the Ripper* poden explotar aquestes debilitats mitjançant atacs de força bruta, de diccionari o contextuals. També s'ha posat de manifest la rellevància de la longitud i complexitat de les contrasenyes per evitar que siguin trencades fàcilment.

En conjunt, la pràctica ofereix una visió completa dels riscos associats a les contrasenyes i dels mecanismes que es poden utilitzar tant per protegir com per atacar sistemes que depenen de la seva seguretat. Aquesta doble perspectiva és essencial per formar professionals amb una visió crítica i global en l'àmbit de la ciberseguretat.

Bibliografia

- Jack, P. B. (s. f.). *Hash Analyzer - TunnelsUP*.

  https://www.tunnelsup.com/hash-analyzer/

- weakpass.com. (n.d.). *rockyou.txt – Common passwords wordlist*. https://weakpass.com/wordlists/rockyou.txt

- colaboradores de Wikipedia. (2022, 19 marzo). *Filtro de Bloom*. Wikipedia, la

  Enciclopedia Libre. https://es.wikipedia.org/wiki/Filtro_de_Bloom

- Academy, B. (2023, 30 mayo). ¿Qué es un Bloom Filter? *Bit2Me Academy*.

  https://academy.bit2me.com/que-es-un-bloom-filter/

- GeeksforGeeks. (2024, 29 marzo). *Double hashing*. GeeksforGeeks.

  https://www.geeksforgeeks.org/double-hashing/

Annexos

- Codi utilitzat en aquesta pràctica: https://github.com/Dakuur/criptografia/tree/main/lab2